Intro to SOLID Principles

Special class

Love Babbar • Mar 13, 2024

SOLID Principles

- Lakshay

Good coder := Quality code

1. Reusable code
2. Extensible
3. Flexible
4. Stable → Exception Handling
5. Reliability
6. Modularity
7. Security
8. Correctness

functional

= ↓

Tree

fun()

↓

fun2()

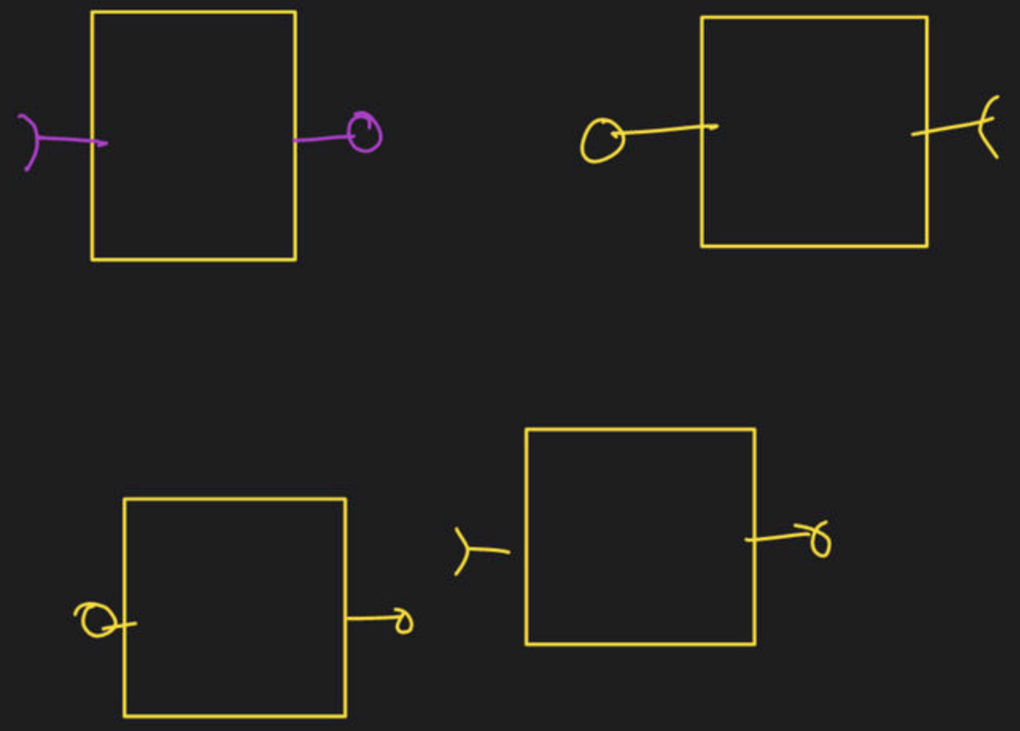
↓

3()

↘ 4()

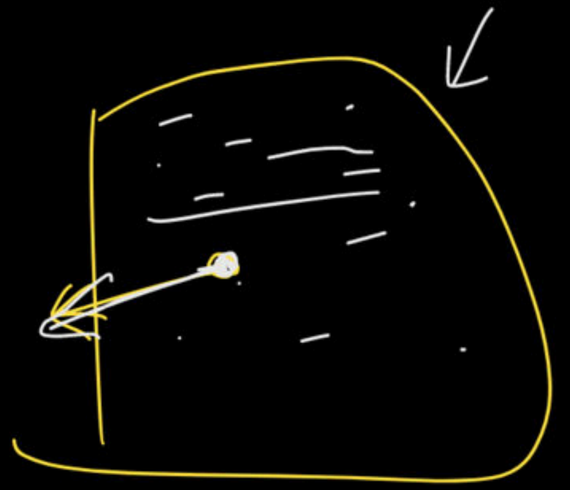
⋮

(LeGo)



Purpose

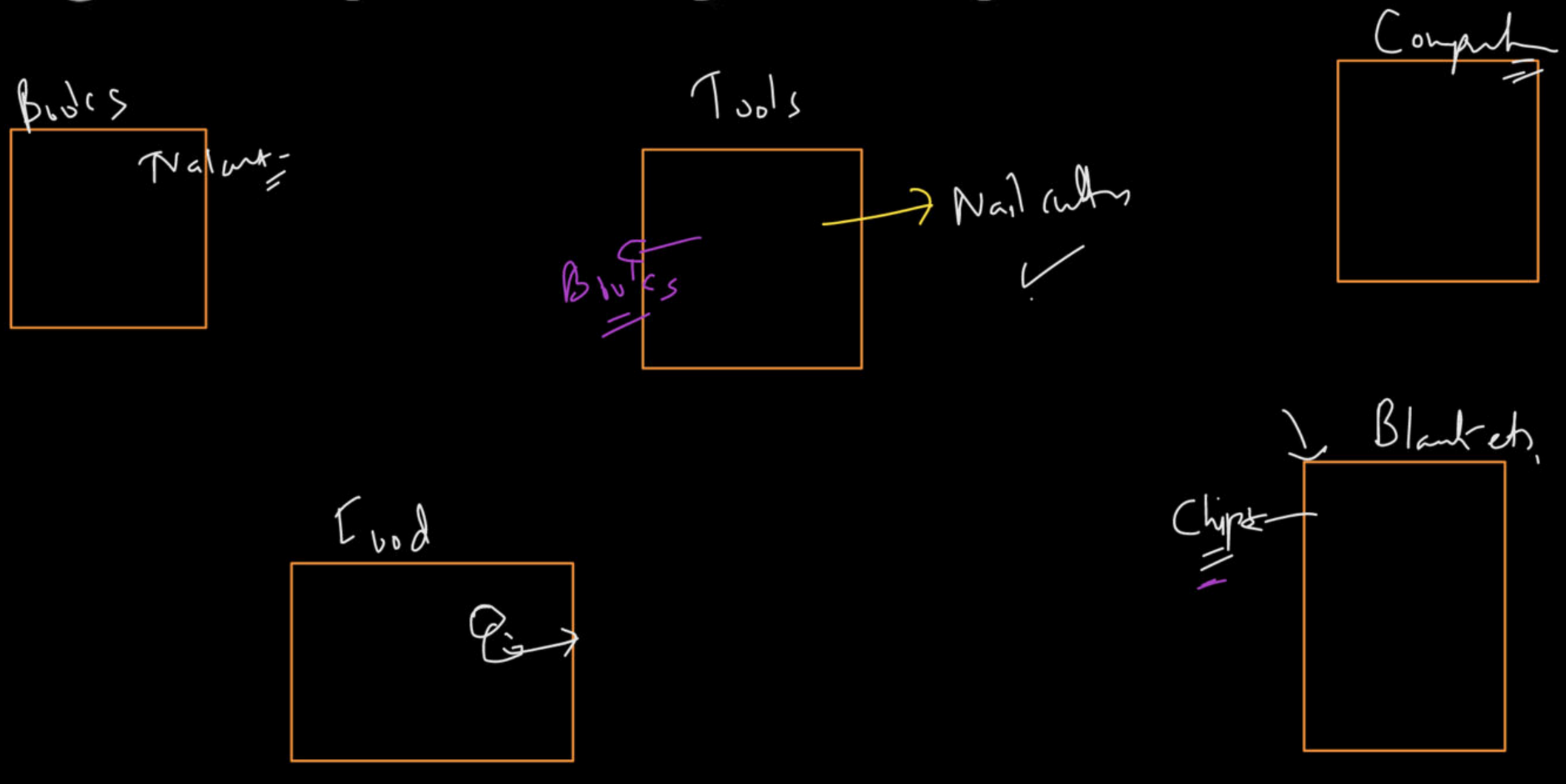
1. Introduced by Robert Martin (Uncle Bob), named by Michael Feathers.
2. To make code more maintainable, easy to reuse.
3. To make it easier to quickly extend the system with new functionality without breaking the existing ones.
4. To make the code easier to read and understand, thus spend less time figuring out what it does and more time actually developing the solution. (Time Saving)



Single Responsibility Principle

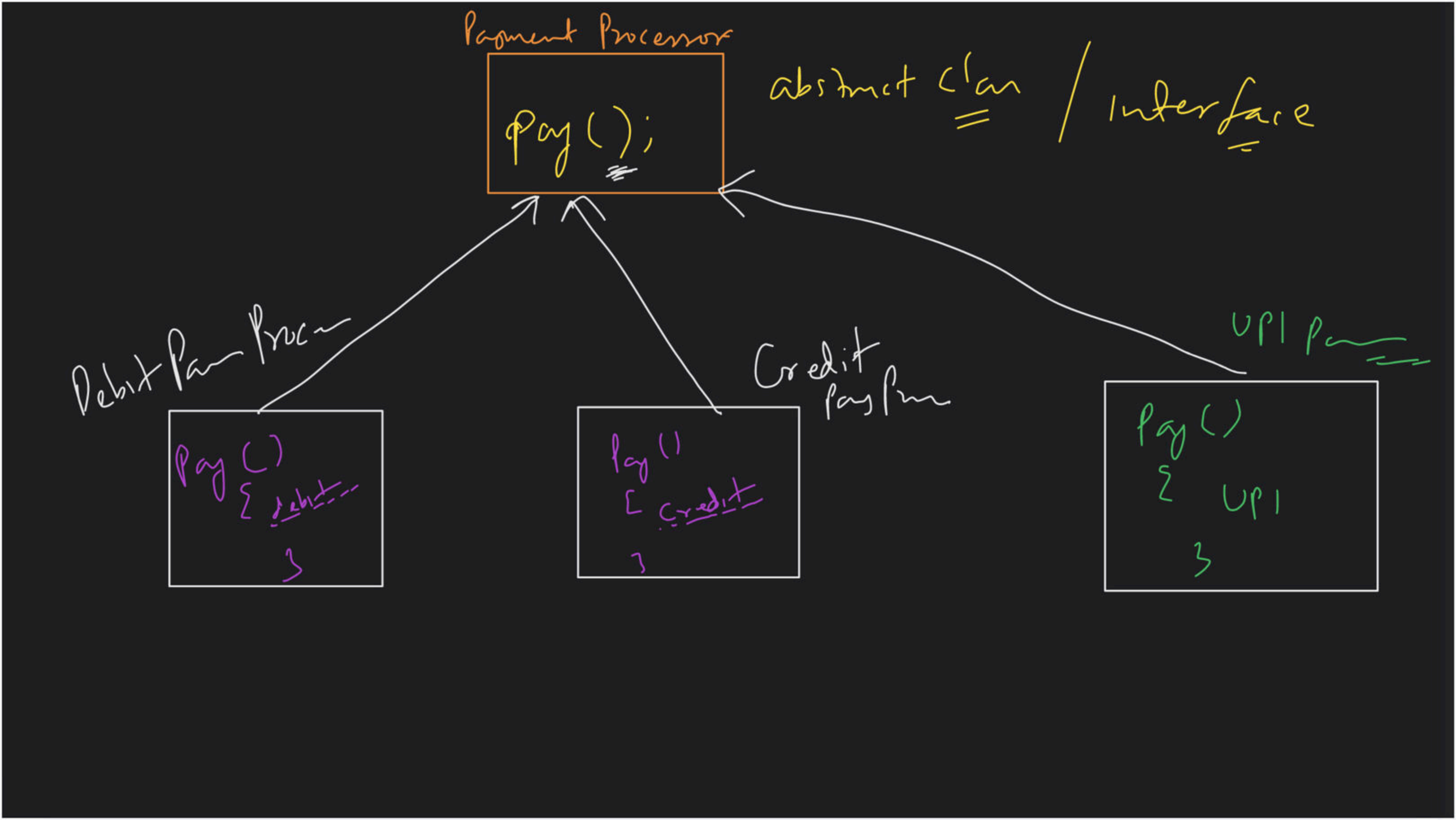
1. A class should have one, and only one reason to change. This means that a class should only have one job or responsibility.
2. A class should only be responsible for one thing.
3. There's a place for everything and everything in its place.
4. Find one reason to change and take everything else out of the class.
5. **Importance**: Following SRP makes your code more modular, easier to understand, maintain, and extend. It helps in isolating functionalities, making debugging and testing more straightforward.

Single Responsibility Principle

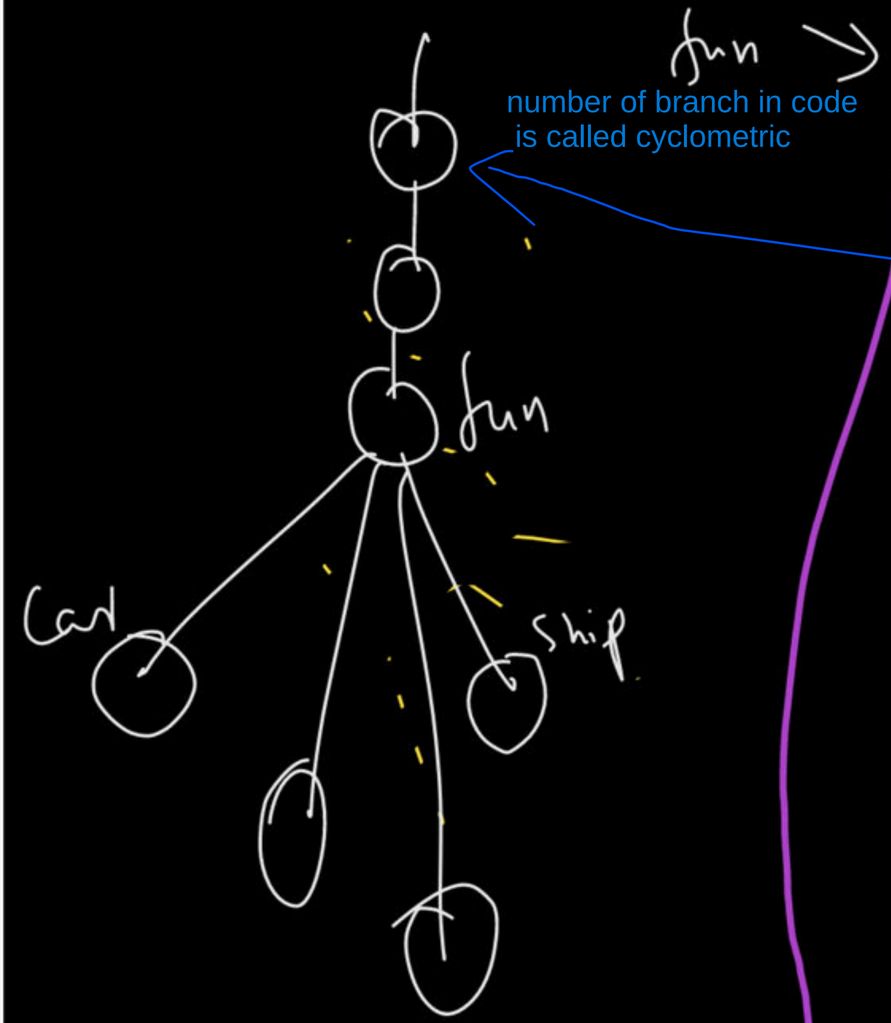


Open-Closed Principle

1. An entity should be open for extension but closed for modification. This means you should be able to add new functionality without changing the existing code.
2. Extend functionality by adding new code instead of changing existing code.
3. **Goal**: Get to a point where you can never break the core of your system.
4. **Importance**: OCP encourages a more stable and resilient codebase. It promotes the use of interfaces and abstract classes to allow for behaviors to be extended without modifying existing code.
5. Writing code structure in such a way new functionality can be added by adding new code not by modifying existing code.


Payment Processor
pay();
abstract class / interface
Debit Pay Proc
pay()
{ debit...
}
Credit Pay Proc
pay()
{ credit
}
UPI Pay
pay()
{ UPI
}

Open-Closed Principle




```
for(Vehicle vehicle : vehicles) {
    switch(vehicle.getType()) {
        case CAR:
            vehicle.lock();
            vehicle.go();
            break;
        case SHIP:
            vehicle.balance();
            vehicle.swim();
            break;
        case AIRPLANE:
            vehicle.go();
            vehicle.fly();
            break;
        case TANK:
            vehicle.move();
            vehicle.stop();
            vehicle.fire();
            break;
    }
    vehicle.stop();
}
```

Bad

1. if, switch
2. Violate OCP
3. Cyclometric Complexity
4. Downcasting
5. Typechecking
   anti - ABS

Open-Closed Principle

```
1    do(Car v){
2        vehicle.lock();
3        vehicle.go();
4    }
5    do(Ship v){
6        vehicle.balance();
7        vehicle.swim();
8    }
9    do(Airplane v){
10       vehicle.go();
11       vehicle.fly();
12   }
13   do(Tank v){
14       vehicle.move();
15       vehicle.stop();
16       vehicle.fire();
17   }
18
19   execute(List<Vehicle> vehicles){
20       for(Vehicle vehicle : vehicles) {
21           do(vehicle);
22           vehicle.stop();
23       }
24   }
```

Compile X
Run Time Error

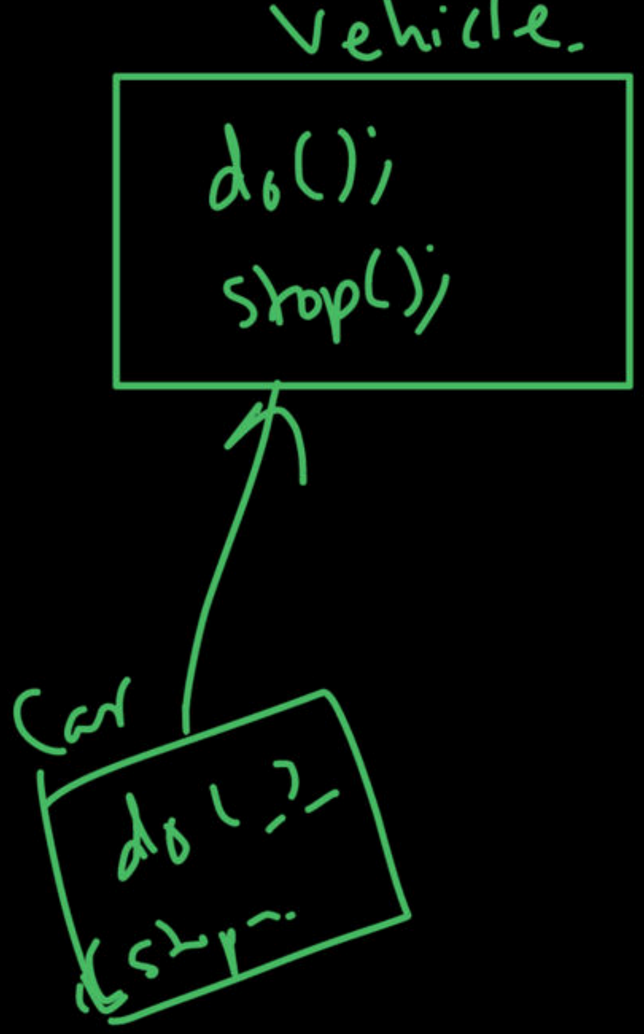
Down casting

Open-Closed Principle

```
1   do(Car vehicle){
2       vehicle.lock();
3       vehicle.go();
4   }
5   do(Ship vehicle){
6       vehicle.balance();
7       vehicle.swim();
8   }
9   do(Airplane vehicle){
10      vehicle.go();
11      vehicle.fly();
12  }
13  do(Tank vehicle){
14      vehicle.move();
15      vehicle.stop();
16      vehicle.fire();
17  }
18
19  execute(List<Vehicle> vehicles){
20      for(Vehicle vehicle : vehicles) {
21          if(vehicle.instanceof(Car))
22              do((Car) vehicle)
23          if(vehicle.instanceof(Tank))
24              do((Tank) vehicle)
25
26          ....
27      }
28  }
```

Downcasting

Open-Closed Principle




```
1   interface Vehicle{
2     do();
3     stop();
4   }
5   class Car implements Vehicle{
6     do(){
7       lock();
8       go();
9     }
10    ...
11  }
12  class Ship implements Vehicle{
13    do(){
14      balance();
15      swim();
16    }
17    ...
18  }
19  class Airplane implements Vehicle{
20    do(){
21      go();
22      fly();
23    }
24    ...
25  }
26  class Tank implements Vehicle{
27    do(){
28      move();
29      stop();
30      fire();
31    }
32    ...
33  }
34
35  ...ute(List<Vehicle> vehicles){
36    for(Vehicle vehicle : vehicles) {
37      vehicle.do();
38      vehicle.stop();
```